Task 5.1 – ShapeDrawer - Multiple shapes

Drawing.cs

```
using System;
using System.Collections.Generic;
using SplashKitSDK;

namespace ShapeDrawer
{

    public class Drawing
    {
        // Private fields
        private readonly List<Shape> _shapes;
        private Color _background;

        // Constructor
        public Drawing(Color background)
        {
            _shapes = new List<Shape>();
            _background = background;
        }

        // Default constructor using Color.White
        public Drawing() : this(Color.White)
        {
            // other steps could go here…
        }

        //Properties
        public List<Shape> SelectedShapes
        {
            get
    {
        List<Shape> result = new List<Shape>();
        foreach (Shape s in _shapes)
        {
            if (s.Selected)
            {
                result.Add(s);
            }
        }
        return result;
    }
}

public int ShapeCount
{
    get { return _shapes.Count; }
}

public Color Background
{
    get { return _background; }
    set { _background = value; }
}

//Methods
public void Draw()
{
    SplashKit.ClearScreen(_background);
    foreach (Shape s in _shapes)
    {
        s.Draw();
    }
}

// SelectShapesAt method that selects/deselects shapes at given point
public void SelectShapesAt(Point2D pt)
```

```
        {
            foreach (Shape s in _shapes)
            {
                if (s.IsAt(pt))
                {
                    s.Selected = true;
                }
                else
                {
                    s.Selected = false;
                }
            }
        }

        public void AddShape(Shape s)
        {
            _shapes.Add(s);
        }

        public void RemoveShape(Shape s)
        {
            _shapes.Remove(s);
        }
    }
}
```

Shape.cs

```
using System;
using SplashKitSDK;

namespace ShapeDrawer;

public class Shape
{
    //Fields
    private Color _color; //changed from string to Color
    private float _x;
    private float _y;
    private int _width;
    private int _height;
    private bool _selected; // Add selected field

    //Default constructor for creating new shapes on the fly
    public Shape()
    {
        _color = Color.Green;
        _x = 0.0f;
        _y = 0.0f;
        _width = 100;
        _height = 100;
        _selected = false;
    }

    //Original constructor
    public Shape(int param)
    {
        _color = Color.Chocolate; // As my name is Min Thu Kyaw Khaung, the first letter 'M' which is
after A-L.
        _x = 0.0f;
        _y = 0.0f;
        _width = param;
    _height = param;
    _selected = false; // Initialize selected to false
}

//Properties
public Color Color
{
    get { return _color; }
    set { _color = value; }
}

public float X
{
    get { return _x; }
    set { _x = value; }
}
public float Y
{
    get { return _y; }
    set { _y = value; }
}
public int Width
{
    get { return _width; }
    set { _width = value; }
}
public int Height
{
    get { return _height; }
    set { _height = value; }
}

// Add a property for selected
public bool Selected
{
    get { return _selected; }
```

```
        set { _selected = value; }
    }

    //Draw the shape
    public void Draw()
    {
        SplashKit.FillRectangle(_color, _x, _y, _width, _height); //changed from Console.WriteLine
statements
        if (_selected) // Draw a border if selected
        {
            DrawOutline();
        }
    }

    //Draw outline around the shape
    public void DrawOutline()
    {
        // The outline is 6 pixels wider on all sides (5 + 1 (Last ID))
        SplashKit.DrawRectangle(Color.Black, _x - 6, _y - 6, _width + 12, _height + 12);
    }

    ///Check if the point is within the shape's bounds
    public bool IsAt(Point2D pt)
    {
        return pt.X >= _x && pt.X <= (_x + _width) &&
                pt.Y >= _y && pt.Y <= (_y + _height);
    }
}
```

Program.cs

```csharp
using System;
using System.Collections.Generic;
using SplashKitSDK;

namespace ShapeDrawer;
    public class Program
    {
        public static void Main()
        {
            Window window = new Window("Shape Drawer - Multiple Shapes", 800, 600);

            // Create a new Drawing object
            Drawing myDrawing = new Drawing();

            do
            {
                SplashKit.ProcessEvents();

                // Check if left mouse button is clicked
                if (SplashKit.MouseClicked(MouseButton.LeftButton))
                {
                    // Create a new Shape object using the default constructor
                    Shape myShape = new Shape(181);

                    // Move the shape to where the mouse was clicked
                    myShape.X = SplashKit.MouseX();
                    myShape.Y = SplashKit.MouseY();

                    // Add the shape to the drawing
                    myDrawing.AddShape(myShape);
                }

                // Check if spacebar is pressed
                if (SplashKit.KeyTyped(KeyCode.SpaceKey))
                {
                    // Change the background color to a new random color
                    myDrawing.Background = SplashKit.RandomColor();
                }

                // Check if right mouse button is clicked
                if (SplashKit.MouseClicked(MouseButton.RightButton))
                {
                    // Get current mouse position
                    Point2D mousePos = SplashKit.MousePosition();
                    // Tell myDrawing to SelectShapesAt the current mouse pointer position
                    myDrawing.SelectShapesAt(mousePos);
                }

                // Check if Delete key or Backspace key is pressed
                if (SplashKit.KeyTyped(KeyCode.DeleteKey) ||
SplashKit.KeyTyped(KeyCode.BackspaceKey))
                {
                    // Get all selected shapes and remove them from the drawing
                    List<Shape> selectedShapes = myDrawing.SelectedShapes;
                    foreach (Shape shape in selectedShapes)
                    {
                        myDrawing.RemoveShape(shape);
                    }
                }

                // Tell myDrawing to Draw
                myDrawing.Draw();

                SplashKit.RefreshScreen();

            } while (!window.CloseRequested);
        }
    }
```

Task 5.2 – SwinAdventure Iteration 4 Inheritance

GameObject.cs

```
using System;
using System.Collections.Generic;

namespace SwinAdventure;

public abstract class GameObject : IdentifiableObject
{
    private string _description;
    private string _name;

    public GameObject(string[] ids, string name, string desc) : base(ids)
    {
        _name = name;
        _description = desc;
    }

    public string Name
    {
        get { return _name; }
    }

    public virtual string ShortDescription
    {
        get { return _name + " (" + FirstId + ")"; }
    }

    public virtual string FullDescription
    {
        get { return _description; }
    }
}
```

Item.cs

```
using System;
using System.Collections.Generic;

namespace SwinAdventure
{
    public class Item : GameObject
    {
        // Constructor for the Item.
        public Item(string[] idents, string name, string desc) : base(idents, name, desc)
        {

        }
    }
}
```

Inventory.cs

```
using System;
using System.Collections.Generic;

namespace SwinAdventure
{
    public class Inventory
    {
        // Fields
        private List<Item> _items;

        //Constructor
        public Inventory()
        {
            _items = new List<Item>();
        }

        //Methods
        public bool HasItem(string id)
        {
            foreach (Item item in _items)
            {
                if (item.AreYou(id))
                {
                    return true;
                }
            }
            return false;
        }

        public void Put(Item itm)
        {
            _items.Add(itm);
        }
```

```
public Item? Take(string id)
{
    for (int i = 0; i < _items.Count; i++)
    {
        if (_items[i].AreYou(id))
        {
            Item item = _items[i];
            _items.RemoveAt(i);
            return item;
        }
    }
    return null;
}

public Item? Fetch(string id)
{
    foreach (Item item in _items)
    {
        if (item.AreYou(id))
        {
            return item;
        }
    }
    return null;
}

//Property
public string ItemList
{
    get
    {
        string result = "";
        foreach (Item item in _items)
        {
            result = result + "\t" + item.ShortDescription + "\n";
        }
```

```
                return result;
            }
        }
    }
}
```

IdentifiableObject.cs

```
using System;
using System.Collections.Generic;

namespace SwinAdventure
{
    public class IdentifiableObject
    {
        //Collection class to store identifiers

        private List<string> _identifiers;

        // Constructor: Initializes the object with an array of identifiers.
        public IdentifiableObject(string[] idents)
        {
            _identifiers = new List<string>();
            foreach (string id in idents)
            {
                AddIdentifier(id);
            }
        }

        // Checks if a given 'id' is in the list (case-insensitive).
        public bool AreYou(string id)
        {
            return _identifiers.Contains(id.ToLower());
        }

        // Add FirstId property
        // Gets the first identifier, or an empty string if the list is empty.
        public string FirstId
        {
            get
            {
                if (_identifiers.Count > 0)
                {
                    return _identifiers[0];
                }
                else
                {
                    return "";
                }
            }
        }

        // AddIdentifier Method
        // Adds a new identifier to the list in lowercase.
        public void AddIdentifier(string id)
        {
            _identifiers.Add(id.ToLower());
        }

        // RemoveIdentifier Method
        // Removes an identifier from the list.
        public void RemoveIdentifier(string id)
        {
            _identifiers.Remove(id.ToLower());
        }

        // PrivilegeEscalation Method
        // Replaces the first ID if the correct PIN is provided.
        public void PrivilegeEscalation(string pin)
        {
            if (pin == "4881" && _identifiers.Count > 0)
            {
                _identifiers[0] = "TUTE01";
            }
        }
    }
}
```

ItemTests.cs

```
using NUnit.Framework;
using SwinAdventure;

namespace SwinAdventure.Tests
{
    [TestFixture]
    public class ItemTests
    {
        private Item _testItem;

        [SetUp]
        public void Setup()
        {
            // Initialize the test item with sample identifiers.
            _testItem = new Item(new string[] { "sword", "bronze sword" }, "bronze sword", "A short sword
cast from bronze");
        }

        [Test]
        public void TestItemIsIdentifiable()
        {
            // Test that item responds correctly to AreYou requests
            Assert.That(_testItem.AreYou("sword"), Is.True);
            Assert.That(_testItem.AreYou("bronze sword"), Is.True);
            Assert.That(_testItem.AreYou("SWORD"), Is.True);
            Assert.That(_testItem.AreYou("axe"), Is.False);
        }

        [Test]
        public void TestShortDescription()
        {
            // Test short description format: "a name (first id)"
            Assert.That(_testItem.ShortDescription, Is.EqualTo("bronze sword (sword)"));
        }
        [Test]
        public void TestFullDescription()
        {
            // Test that full description returns the item's description
            Assert.That(_testItem.FullDescription, Is.EqualTo("A short sword cast from bronze"));
        }

        [Test]
        public void TestPrivilegeEscalation()
        {
            // Test privilege escalation with correct PIN
            _testItem.PrivilegeEscalation("4881");
            Assert.That(_testItem.FirstId, Is.EqualTo("TUTE01"));
        }
    }
}
```

InventoryTests.cs

```
using NUnit.Framework;
using SwinAdventure;

namespace SwinAdventure.Tests
{
    [TestFixture]
    public class InventoryTests
    {
        private Inventory _inventory;
        private Item _testItem1;
        private Item _testItem2;

        [SetUp]
        public void Setup()
        {
            _inventory = new Inventory();
            _testItem1 = new Item(new string[] { "sword", "axe" }, "bronze sword", "A basic bronze sword");
            _testItem2 = new Item(new string[] { "gem", "ruby" }, "red gem", "A shiny red ruby");
        }

        [Test] //The Inventory has items that are put in it.
        public void TestFindItem()
        {
            // Arrange
            _inventory.Put(_testItem1);
            _inventory.Put(_testItem2);

            // Act & Assert
            Assert.That(_inventory.HasItem("sword"), Is.True, "Should find sword in inventory");
            Assert.That(_inventory.HasItem("axe"), Is.True, "Should find weapon identifier for sword");
            Assert.That(_inventory.HasItem("gem"), Is.True, "Should find gem in inventory");
            Assert.That(_inventory.HasItem("ruby"), Is.True, "Should find ruby identifier for gem");
        }
        [Test] //The Inventory does not have items it does not contain.
        public void TestNoItemFind()
        {
            // Arrange
            _inventory.Put(_testItem1);

            // Act & Assert
            Assert.That(_inventory.HasItem("shield"), Is.False, "Should not find shield in inventory");
            Assert.That(_inventory.HasItem("potion"), Is.False, "Should not find potion in inventory");
            Assert.That(_inventory.HasItem("gold"), Is.False, "Should not find gem when not in
inventory");
        }

        [Test] //Returns items it has, and the item remains in the inventory.
        public void TestFetchItem()
        {
            // Arrange
            _inventory.Put(_testItem1);
            _inventory.Put(_testItem2);

            // Act
            Item? fetchedSword = _inventory.Fetch("sword");
            Item? fetchedGem = _inventory.Fetch("gem");

            // Assert
            Assert.That(fetchedSword, Is.Not.Null, "Should return a valid item");
            Assert.That(fetchedGem, Is.Not.Null, "Should return a valid item");
            Assert.That(fetchedSword, Is.SameAs(_testItem1), "Should return the same sword item");
            Assert.That(fetchedGem, Is.SameAs(_testItem2), "Should return the same gem item");

            // Verify items are still in inventory after fetch
            Assert.That(_inventory.HasItem("sword"), Is.True, "Sword should still be in inventory after
fetch");
            Assert.That(_inventory.HasItem("gem"), Is.True, "Gem should still be in inventory after fetch");
        }
        [Test] // Returns the item, and the item is no longer in the inventory.
        public void TestTakeItem()
        {
            // Arrange
            _inventory.Put(_testItem1);
            _inventory.Put(_testItem2);

            // Act
            Item? takenSword = _inventory.Take("sword");
            Item? takenGem = _inventory.Take("gem");

            // Assert
            Assert.That(takenSword, Is.Not.Null, "Should return a valid item");
            Assert.That(takenGem, Is.Not.Null, "Should return a valid item");
            Assert.That(takenSword, Is.SameAs(_testItem1), "Should return the same sword item");
            Assert.That(takenGem, Is.SameAs(_testItem2), "Should return the same gem item");

            // Verify item is no longer in inventory after take
            Assert.That(_inventory.HasItem("sword"), Is.False, "Sword should not be in inventory after
take");
            Assert.That(_inventory.HasItem("gem"), Is.False, "Gem should not be in inventory after take");
        }

        [Test] //Returns a string containing multiple lines. Each line contains a tab-indented short
description of an item in the Inventory.
        public void TestItemList()
        {
            // Arrange
            _inventory.Put(_testItem1);
            _inventory.Put(_testItem2);

            // Act
            string itemList = _inventory.ItemList;

            // Assert
            Assert.That(itemList.Contains("\tbronze sword (sword)"), Is.True, "Item list should contain
tabbed sword description");
            Assert.That(itemList.Contains("\tred gem (gem)"), Is.True, "Item list should contain tabbed
gem description");
        }
    }
}
```